Automatic Parallelization Tool Efficiency: FPP/Atexpert Case Studies

Robert J. Bergeron

Report RND-93-011 June 1993

NAS Systems Development Branch
NAS Systems Division
NASA Ames Research Center
Mail Stop 258-6
Moffett Field, CA 94035-1000

Automatic Parallelization Tool Efficiency: FPP/Atexpert Case Studies

Robert J. Bergeron
Computer Sciences Corporation
NASA Ames Research Center
Moffett Field, CA 94035, USA

Abstract

This paper presents several synthetic Fortran codes for testing the ability of automatic parallelizers to generate efficient parallel code. The codes typify sequential solution techniques currently in use at NAS and represent the types of algorithms that automatic parallelizers will be required to analyze and decompose. The Cray automatic parallelizer FPP performed well on codes containing explicitly parallel algorithms, but required user intervention to assist in vectorized relaxation schemes. The inability of FPP to expose parallelism at a level higher than loop level reduced the parallel performance of relaxation and ADI schemes. The Cray utility *atexpert* consistently diagnosed the causes for poorer-than-expected parallel performance.

1.0 Introduction

The current limits on technology indicate that future high speed computer systems will employ multiple processors, operating in parallel, to solve computationally-intensive problems. Efficient utilization of such machines will require a recasting of current highly vectorized programs into highly parallel programs. The past success of automatic vectorization has motivated the development of automatic parallelization tools to reduce the user effort in constructing parallel programs from vectorized programs. Automatic parallelization in the current context means the rewriting of a serial Fortran program for execution on multiple processors using only directives inserted into the code before compilation. Automatic parallelization should help users to construct code for task allocation, CPU synchronization, and interprocessor communication. Such constructs can apply to a coarse-grained high-level partition, corresponding to subroutine-sized tasks, or to a fine-grained, low-level partition, corresponding to loop-sized tasks.

Analysts at the NASA Ames Research Center's Numerical Aerodynamics Simulation (NAS) Facility have been evaluating automatic parallelization tools for some time, beginning with the evaluation of loop-level tools on shared-memory machines (Chen and Pase, 1991). Because these tools seek only to enable the parallel execution of DO-loops, the source program retains its logical structure. Processors executing loop-level parallel work synchronize at the end of the DO-loop and perform a relatively small amount of work between synchronization points. The limited amount of work performed between synchronization constitutes fine-grained parallelism.

Evaluation of automatic tools for highly parallel machines has not yet occurred. Highly parallel machines have distributed memories and require coarser-grained parallel execution to amortize communication overhead. Coarser-grained parallel execution requires substantial rewriting of source programs to enable independent execution of tasks larger than those performed by DO-loops. Code generated by automatic tools for this purpose can assume many forms, depending upon the target parallel architecture and the transformation rules internal to the tool. Evaluation of coarse-grained tools needs at least one example of a manual source transformation for comparison against tool-generated code.

1.1 The Parallel Suite

This report presents the results of automatic loop-level transformations on a suite of five Fortran codes executing on a shared memory machine. The codes, employing vectorized numerical methods, typify solution techniques which NAS users might submit for parallelization by an automatic preprocessor. A subsequent report will present the results of higher-level transformations on these same sources for execution on a distributed memory machine.

The parallel suite includes the following kernels:

- Successive OverRelaxation (SOR) algorithm in a cube geometry,
- PARAllel Cyclic Reduction algorithm (PARACR) solving a two-dimensional tridiagonal system,
- Alternating Direction Implicit (ADI) algorithm in a planar geometry,
- MultiGRid (MGR) algorithm in a cube geometry, and
- Shallow Water Model (SWM) providing an explicit solution to the two-dimensional mass and momentum equations treating wave motion.

The numerical solution of partial differential equations provides the basis for three of the algorithms in the suite: SOR, ADI, and MGR. The basic problem, the solution of Poisson's equation, admits a variety of highly parallel solution algorithms. The second algorithm, PARACR, employs the cyclic reduction method to solve the tridiagonal system arising from the finite-difference approximation to Poisson’s equation. A modification (Hockney and Jesshope, 1988) to the standard odd-even algorithm increases the parallelism by performing all phases of the reduction in parallel. The fifth algorithm, SWM, provides the solution to the mass and momentum equations on a two-dimensional, Cartesian grid. The shallow water algorithm performs the same set of operations on each of the grid points and has no conditional statements.

Problem sizes for the base case versions reflect a grid containing about 1 million points. While this size has been a historical standard for NAS computational fluid dynamics codes (Peterson and Balhaus, 1987), current problems seem to be about a factor of three larger and future problems will be even larger. Accordingly, the report also provides performance data for a case eight times as large as the base case to assess the effects of size on parallel performance.

1.2 The Cray Environment

All calculations reported here were performed on the NAS Cray Y-MP, a 6 nanosecond 8-CPU 256 MW vector computer. The CPUs of the Y-MP are tightly coupled through a shared main memory and sets of shared registers. Loop-level parallelism exploits the high-speed communication available with these shared registers to achieve parallelism at the DO-loop level (Cray, 1991).

Cray's automatic parallelization tool, the Fortran PreProcessor (FPP), performs the automatic distribution of loop iterations to multiple processors using the original Fortran source as input. FPP transforms loop structures into parallel regions by prefixing the structures with loop work distribution commands such as "DO ALL PARALLEL". FPP also prefixes the structures with directives indicating the scope of loop variables, *i.e.*, statements indicating whether a variable can be shared among the processors or whether the variable must be used only by the CPU performing a particular loop iteration. An implicit synchronization occurs at the end of all parallel code regions to allow the multiple CPUs proper access to the shared memory. FPP also provides some interprocedural parallelization capability through its expansion of user-selected subroutines. To facilitate FPP's recognition of data dependencies and its analysis of loop subscripting, the codes in the suite, while performing efficiently on the Y-MP, have been only weakly optimized. FPP version 5.0, release 3.03M3 in conjunction with CFT77, version 5.0, created the executables for the first four codes. An earlier version of FPP, version 4.0, release 2.26B3, created the SWM executables.

Cray also provides the *atexpert* utility (Cray, 1990) to help users improve parallel performance. This tool indicates how the system executes the serial and parallel regions of the FPP-generated code. *Atexpert* provides performance and overhead measurements at the program, subroutine and loop-level. The graphical output format employed by *atexpert* allows the user to obtain a clear picture of parallel performance.

The following sections discuss the technique for evaluating effectiveness of automatic tools, the performance of each of the five codes, and the performance of FPP. Since the codes contain extensive comments describing the actual implementation of the algorithm into Fortran, the discussion of the individual codes emphasizes the parallelism contained therein and an evaluation of the Cray automatic parallelization tool.

2.0 Definitions and Evaluation Procedure

The following section describes some of the terms used to measure parallelism and provides a basis for comparing the source code parallelism with the parallelism extracted by FPP.

2.1 Basic Definitions

The figure of merit for parallel performance is speedup, defined as the ratio of the elapsed time for the code executing on one CPU to the elapsed time for the code executing on a specific number of CPUs. Since a major purpose of the report is FPP evaluation, comparisons will be made using the FPP-generated code run on one CPU and N CPUs. As the original code does not provide the basis for single CPU timings, these comparisons made on this basis do not reflect algorithmic or hardware performance (Bailey, 1992). The performance of these codes should not be taken as measures of the parallel speedup over the best Y-MP implementation.

The efficiency, defined as the speedup divided by the number of CPUs which execute the problem, measures how effectively the program utilizes the hardware; an efficiency close to unity indicates a very effective use of computing resources. The efficiency generally tends to decrease as more processors are brought to bear on a given problem due to contention for shared resources, extra time required to communicate between processors, and the inability of practical algorithms to keep an arbitrary number of processors profitably busy. Parallel efficiencies as low as 0.7 have been quoted as adequate in the literature (Cvetanovic *et al.*, 1990).

2.2 Source Parallelism

A convenient measure of source parallelism involves the assumption that all vectorized operations can be executed in parallel. This measure is strongly hardware dependent because implementations of CPU synchronization and communication will determine the minimum amount of work able to benefit from parallel execution. The Y-MP architecture implements synchronization and communication efficiently and many vector loops will benefit from parallel execution.

Efficient transformation of vector operations into parallel operations requires the vendor's software to expose those vector constructs which can benefit from parallel execution on the vendor's hardware. Some architectures may execute loops in vector mode on one CPU more efficiently than on N CPUs. FPP has internal criteria which select only those loops benefitting from parallel execution.

The assumption that all vectorized operations can be parallelized allows an expression for the relationship between the speedup and the measured vectorization. Let **fv** denote the fraction of program operations (including memory) executed in vector mode and **fs** denote the fraction of operations executed in scalar mode. The maximum speedup, **S**, will be:

(1)

$$S = 1/(fs + fv/NCPU)$$

This maximum speedup calculation, based on the source parallelism contained in the vectorized work, assumes that all vector loops execute in parallel. This calculation assumes that the non-vector, *i.e.*, scalar, work cannot be parallelized. In this report, hardware operation counts will provide **fv** and wall clock timings will provide a measured speedup. If measured speedup is close to the maximum speedup, then FPP will have done a very good job of automatic parallelization.

FPP will try to optimize a code before parallelizing it and sometimes the optimizations and additional scalar code introduced by FPP will change the above fractions. Therefore, the execution time of the parallel code generated by FPP on one CPU provided the basis for determining the fraction of vector and scalar code. This measurement excludes some of the overhead associated with parallelism, *i.e.*, the effort spent testing for permission to execute DO-loops on multiple CPUs.

Architectures without vector hardware may require runtime profiling to obtain the amounts of time spent in single processor execution and parallel execution for comparison of source parallelism and generated parallelism.

2.3 Generated Parallelism

The parallel performance of a code produced with an automatic parallelization tool can help to determine the amount of parallelism generated by the tool. A simple measure of the fraction of code, **f**, executed in serial mode can be defined implicitly as (Karp and Flatt, 1989):

(2)

$$T(p) = T(1) \times f + ((T(1) \times (1-f))/p)$$

where **T(p)** denotes the elapsed time on **p** processors and **T(1)** denotes the elapsed time on one processor. The parallel fraction is **1-f**. This formula assumes that the operating system can distribute an equal amount of work to each of the available processors, *i.e.* , perfect load balancing. The formula also neglects the amount of CPU time required for synchronization. Application of the formula to a vector architecture requires that the vector lengths be sufficiently long to maintain vector performance after the decomposition of DO-loops reduces these lengths by a factor of NCPUs. The report will discuss this aspect and also employ measured single and multiple processor elapsed times to calculate the parallel and serial code fractions.

Thus (1) gives a relation between the measured source parallelism and the maximum, or predicted, speedup and (2) gives a relation between the tool-generated parallelism and the measured, or effective, speedup. Comparison of the predicted parallelism to the effective parallelism will provide an estimate of tool effectiveness.

2.4 Hardware Measurements

The Cray Hardware Performance Monitor (Cray, 1990) measures several quantities which help to explain code performance. One of these is the hardware vector length. The Cray CPU processes vectors in lengths of 64, which is the maximum hardware length for vector instructions. The Hardware Performance Monitor (HPM) reports a weighted average vector length; vector lengths approaching 64 indicate efficient use of the vector units. Another term, “program vector length”, denotes the number of iterations in the dominant vector loops for each program.

The HPM also provides rate of both floating point operations and memory operations. For a given algorithm, the ratio of these two quantities indicates the magnitude of the overhead in terms of memory accesses.

3.0 Successive OverRelaxation Algorithm

3.1 Code Description

NAS users employ the Successive OverRelaxation (SOR) algorithm in advanced formulations involving fluid dynamics on unstructured grids and the aerodynamics of the Space Shuttle (NASA, 1990). The SOR is a Gauss-Seidel iteration which must visit the grid points in a sequential order. The new grid point value, computed as the average of the values at surrounding grid points, replaces the old value as soon as it is computed. The immediate replacement creates a problem for vectorization and parallelization because the iterations of the DO-loop performing the calculation are recursive. (An iteration of a DO-loop is recursive if it has a data dependency on the previous iteration.) Red/black ordering of grid points in a checkerboard fashion (Lambiotte, 1975) removes the recursion because the red updates can be performed independently of the black updates. Red/black ordering is a specific example of “multicolor” ordering (Adams and Ortega, 1975) which promotes parallel finite difference solutions by decoupling the local unknowns.

In addition to vectorization, the red/black algorithm also permits a Chebyshev sequence of relaxation factors to accelerate the convergence of the solution. Features enhancing parallelism include the red/black coloring of the nodes and the data parallelism of the algorithm. The problem geometry and boundary conditions also contribute to the parallelism by allowing application of the same operations to points adjacent to the boundary. Termination occurs when the calculation has reduced the norm of the residual below an input-specified fraction of the initial residual. In this case, the residual is defined as the difference between the discrete solution and the exact solution at each point and the norm is defined as the sum of the absolute values of the residuals.

3.2 Code Performance

This section discusses the singletasked performance of the SOR on several problem sizes and the parallel performance of the base case. The goal in developing a parallel program on a vector supercomputer is to distribute vector work to the various CPUs with a minimum amount of time spent in processor synchronization. The standard SOR formulation does not vectorize easily and the suite version is a red/black SOR in which the outer loop visits all planes as the inner loop visits first the red nodes and then the black nodes. Vectorization of the red/black algorithm requires indirect addressing since the procedure does not visit the nodes comprising the plane in a sequential order.

Table 1 shows singletasked performance as a function of problem size. For the base case, a cube with 128 nodes per edge, the HPM indicated vector lengths of 63 and this value denotes effective use of the vector processor. The SOR base case has a program vector length of 126. Because the SOR accesses arrays containing the geometry coefficients, it displays a somewhat low FLOP-to-memory access ratio of 0.65. Replacement of these coefficients with constant values would improve performance by decreasing memory accesses. The code executes the 8.6 Megaword (MW) base case in 25 seconds, performing at 165 MFLOPS.

Table 1
Singletasked Y-MP SOR Performance

Nodes/Edge	MW	MFLOPS	CPU seconds
32	0.2	140.6	0.1
64	1.2	156.0	1.6
100	4.2	161 0	9.4
128	8.6	165.0	25.0
256	67.3	172.3	289.1

The table shows a modest increase in vector performance with increasing number of nodes as the computationally efficient iteration loops dominate execution time.

With some user assistance, FPP constructed a parallel version from the vector version described above by constructing red and black parallel outer loops. The following fragment illustrates the FPP construction:

```
CMIC@ PARALLEL SHARED(NRLAST, ANORM,...
      R1S=0
CMIC@ DO PARALLEL
       DO 250 K = 1,254
        N=128
c
cc      visit all internal red nodes-bottom to top
c
        DO 100 I=1,NRLAST
         RESID=    RA(N+I,1+K)*B(NBW(N+I),1+K)
     2          + RB(N+I,1+K)*B(NBN(N+I),1+K)
     3          + RC(N+I,1+K)*B(NBE(N+I),1+K)
     4          + RD(N+I,1+K)*B(NBS(N+I),1+K)
     5          + RE(N+I,1+K)*B(N+I,2+K)
     6          + RF(N+I,1+K)*B(N+I,K)
     7          + RG(N+I,1+K)*R(N+I,1+K)
     8          - RH(N+I,1+K)
         R1S=R1S + ABS@(RESID)
         R(N+I,1+K)=R(N+I,1+K)-
     2              W(ITERR)*RESID*RRG(N+I,1+K)
  100 CONTINUE
  250 CONTINUE
CMIC@ GUARD
      ANORM=ANORM+R1S
CMIC@ ENDGUARD
CMIC@ END DO
```

The red outer loop visits all planes, employing an inner loop to update only the red nodes in the plane. Parallel calculation of the red nodes is possible because the red update uses only the black nodes, which themselves remain constant during the red update. After updating its red plane, each CPU enters a critical region (a region admitting only one processor at a time) to make its contribution to the global residual. The CPU then returns to update another red plane. CPU synchronization occurs after the final red update. The parallel black update and black critical region follow the final red critical region. The release of the 5.0 version of FPP employed for the SOR, 3.0.3M5, was unable to construct a parallel loop to obtain global maximum values, but recent releases of FPP have remedied this deficiency.

Table 2 shows base case parallel performance on increasing numbers of CPUs. Section 2.1 defines speedup, efficiency, and serial fraction.

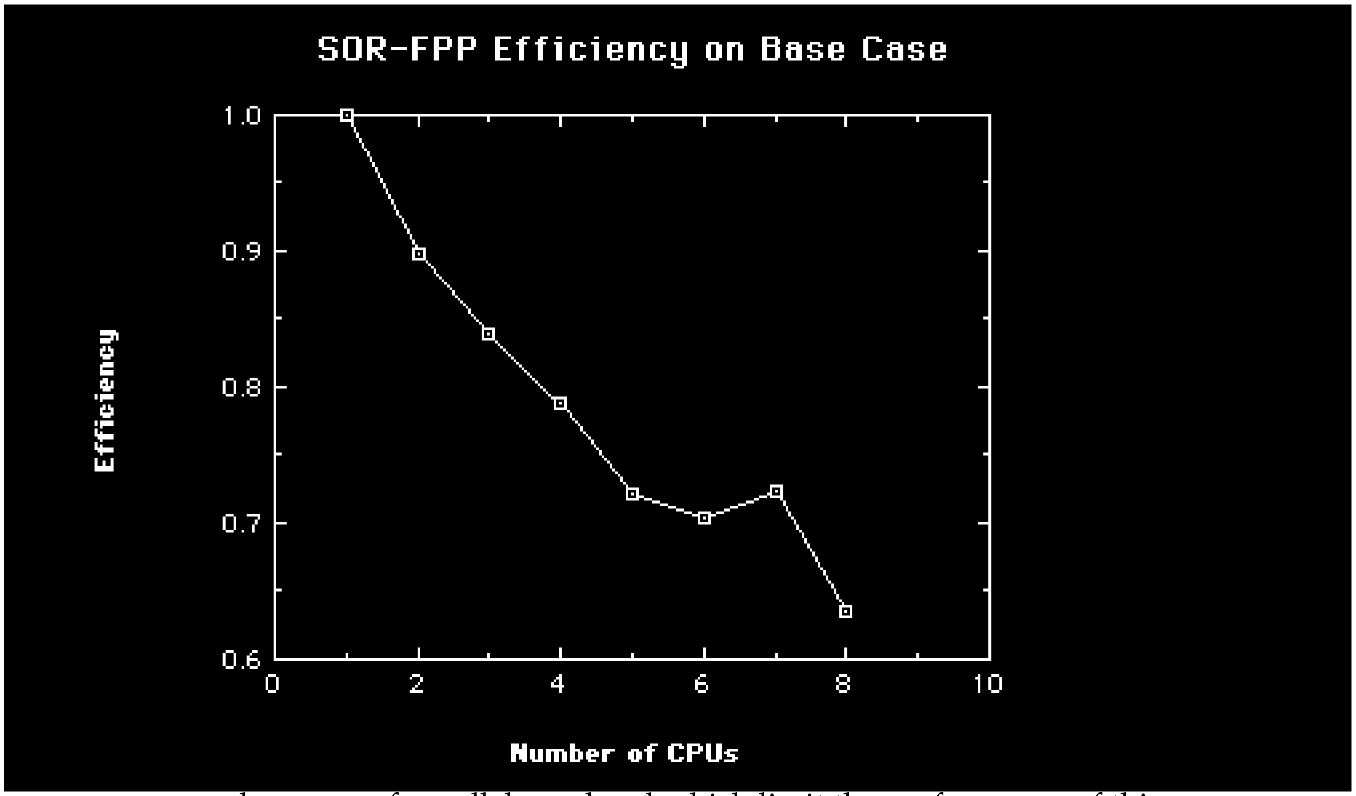
Table 2
Parallel 8-CPU Y-MP Performance of Base Case SOR (128**3)

NCPUS	Elapsed Time	Speedup	Efficiency	Serial Fraction
1	25.076	1.000	1.000	1.000
2	13.954	1.797	0.899	0.139
3	9.950	2.520	0.840	0.095
4	7.943	3.607	0.789	0.089
5	6.952	3.524	0.721	0.097
6	5.942	4.220	0.703	0.084
7	4.948	5.068	0.724	0.064
8	5.082	5.085	0.636	0.082

Figure 1 shows that the efficiency tends to decrease less at NCPUs equal to 3 and 6 and even improves at NCPUS equal to 7. FPP recognized that the base case performs 126 outer iterations and rewrote the code accordingly. Since the number of outer iterations is divisible by 3, 6, and 7, the CPUs can execute an integral number of outer loop iterations, thus perfectly balancing the computational workload.

Figure 1

Examination of code performance with the Cray utility *atxpert* indicated



several sources of parallel overhead which limit the performance of this SOR on increasing numbers of CPUs. The algorithm employs a global sum to determine the convergence of the solution and this approach requires a region of code accessible to one processor at a time. FPP creates a critical region, *i.e.*, a region of code which may be executed by only one CPU at a time; this region generates overhead through its synchronization at the end of the red and black loops.

In the 8-CPU base case, *atexpert* indicated additional overhead due to CPU load imbalance. For this number of CPUs, the total work as represented by the 126 outer iterations of the DO-loops does not parcel into equal work for all CPUs and some CPUs idle for lack of work. *Atexpert* also indicated a second source of overhead involving memory contention since the indirect addressing required by vectorization creates nonsequential memory access patterns.

3.3 FPP Evaluation

This section discusses the performance of FPP by showing, in Table 3, the speedup of the 8-CPU SOR as a function of problem size. The column "Pct Vector" provides the percent of code vectorized by the compiler as calculated from HPM data. Use of this data in conjunction with equation (1) presented in Section 2.1 allows an estimate of the maximum speedup denoted in the table by "Smax". The column labelled "Smeas" denotes the measured speedup and the column labelled "Pct Parallel" represents the amount of FPP-generated parallel code as obtained by application of equation (2) as given in Section 2.1

Table 3
Parallel Y-MP Performance of the SOR

Cube Size	Pct Vector	Smax	Smeas	Pct Parallel
32	95.44	6.06	1.42	36.57
64	96.02	6.26	4.72	90.08
100	96.94	6.59	4.85	91.32
128	97.04	6.63	4.97	91.81
256	97.14	6.67	6.61	97.00

The difference between the maximum speedup based on percent of vector operations and the measured speedup based on wall clock time diminished as the system size increased. The measured performance of the code increased dramatically with problem size. Relative to the 128 node cube, *atexpert* indicated that the largest cube size incurred smaller overhead from load imbalance and reduced memory conflict delays. Table 3 indicates that FPP has extracted essentially all of the parallelism available at a cube size of 256 and FPP's version of the code performs at over 80% efficiency. For cube sizes exceeding 64, decomposition of the SOR at the DO-loop level proved to be an efficient strategy for achieving parallelism.

4.0 Tridiagonal Solver—Parallel Cyclic Reduction

4.1 Code Description

Tridiagonal systems occur repeatedly in finite-difference approximations to partial differential equations with 2nd-order derivatives. Many of the efficient numerical algorithms solving such systems employ some variant of Gaussian reduction and require serial execution.The serial version of the tridiagonal solver in this suite is called the standard (odd-even) cyclic reduction technique (Buzbee, *et al.*, 1970) and the parallel version is termed the parallel cyclic reduction technique (Hockney and Jesshope, 1988). This algorithm is a highly effective method for solving tridiagonal systems, both on a vector computer and on a multiprocessor (Kumar, 1989). This algorithm, arising from the solution of a 5-point finite difference operator, allows the ADI code discussed in the next section to employ this procedure as its solver.

The standard cyclic reduction method consists of iterations, each of which eliminates the odd-numbered equations remaining in the system. The procedure finally arrives at a single equation; the algorithm then backsubstitutes through the reduced systems until the original set of equations is solved. The algorithm consists of 5 loops. The first loop clears (zeroes) the temporary arrays and a second loop initializes these arrays with problem data. These are followed by an outer loop which executes until it has reduced the system to a single equation; its inner loop computes all the recurrence relations needed to reduce the system by a factor of two. The final loop performs the backsubstitution. The version in the suite allows parallel execution of the backsubstitution and requires more floating point operations than the odd-even cyclic reduction.

While the version in this suite requires that the matrix dimension be a power of 2, more complicated algorithms (Sweet, 1974) do not require this restriction. Moreover, a cutoff method (Vu and Yang, 1988) provides a practical approach for relaxing the requirement on the current algorithm and extending the approach to all matrix dimensions. Each cycle of the reduction will decrease the number of independent equations by a factor of two until the number of equations falls below a user-specified value. At this point, the cutoff method solves the remaining system with a Gaussian solver. The cutoff method then proceeds with a back substitution exactly as in the power-of-two method.

The cyclic reduction algorithm differs from the block tridiagonal solver used in many NAS CFD codes. The cyclic reduction algorithm solves a system with one governing equation, whereas the CFD codes treat systems with several governing equations. The CFD block solver employs a Gaussian elimination algorithm to solve the system and a Cholesky decomposition to solve each block (Pulliam and Chausee, 1980). While three-dimensional versions of this algorithm perform well in vector and parallel mode (Bailey *et al.*, 1991), two-dimensional versions display poor parallel performance. Extension of the cyclic reduction method to the block solvers arising in two-dimensional CFD problems should improve their parallel performance.

4.2 Code Performance

This section discusses the singletasked performance of the PARACR and the parallel performance of the base case.

Table 4 shows singletasked performance as a function of matrix size. The base case treats a 2**16 linear system, *i.e.*, a system of 65536 coupled equations. (For comparison, a typical two-dimensional CFD problem, 100 by 100 with 4 equations per node, has about 40,000 unknowns.) The code executes the 7.3 MW base case in 18 seconds, performing at 126 MFLOPS. The HPM indicated vector lengths exceeding 63 and a FLOP-to-memory access ratio of 0.94. The long vector length means that the vector startup overhead is low and the ratio of FLOP to memory access near 1.0 means that the algorithm is computationally efficient. For 100 passes through the solver, single processor performance data are as follows:

Table 4
Singletasked Y-MP PARACR Performance

System Size	MW	MFLOPS	CPU seconds
2**14	1.7	117.5	4.2
2**15	3.5	124.6	8.4
2**16	7.3	125.9	17.7
2**17	15.3	122.8	38.4
2**18	32.1	126.4	78.8
2**19	67.2	127.2	165.0

The table shows only a small increase in vector performance with system size. The algorithm contains an amount of scalar indexing which increases with system size, and perhaps user optimization to reduce this indexing work would improve code performance.

The loop structure of the parallel program constructed by FPP followed the same order as the vector version described above. FPP parallelized the initialization loop and arranged for parallel execution of the inner loop iterations. FPP distributed the inner loop iterations to the processors in groups of 64 to maximize CPU vector performance. The clearing loop remained singletasked because it contained insufficient work to overcome the overhead from parallel execution. The final loop, backsubstitution, also executed in parallel. Processor synchronization occurs at the end of the initialization loop, after each of the inner loops, and at the end of the backfilling loop. In this algorithm, the number of synchronizations depends on the number of iterations required to reduce the system; the base case requires 21 synchronizations per system.

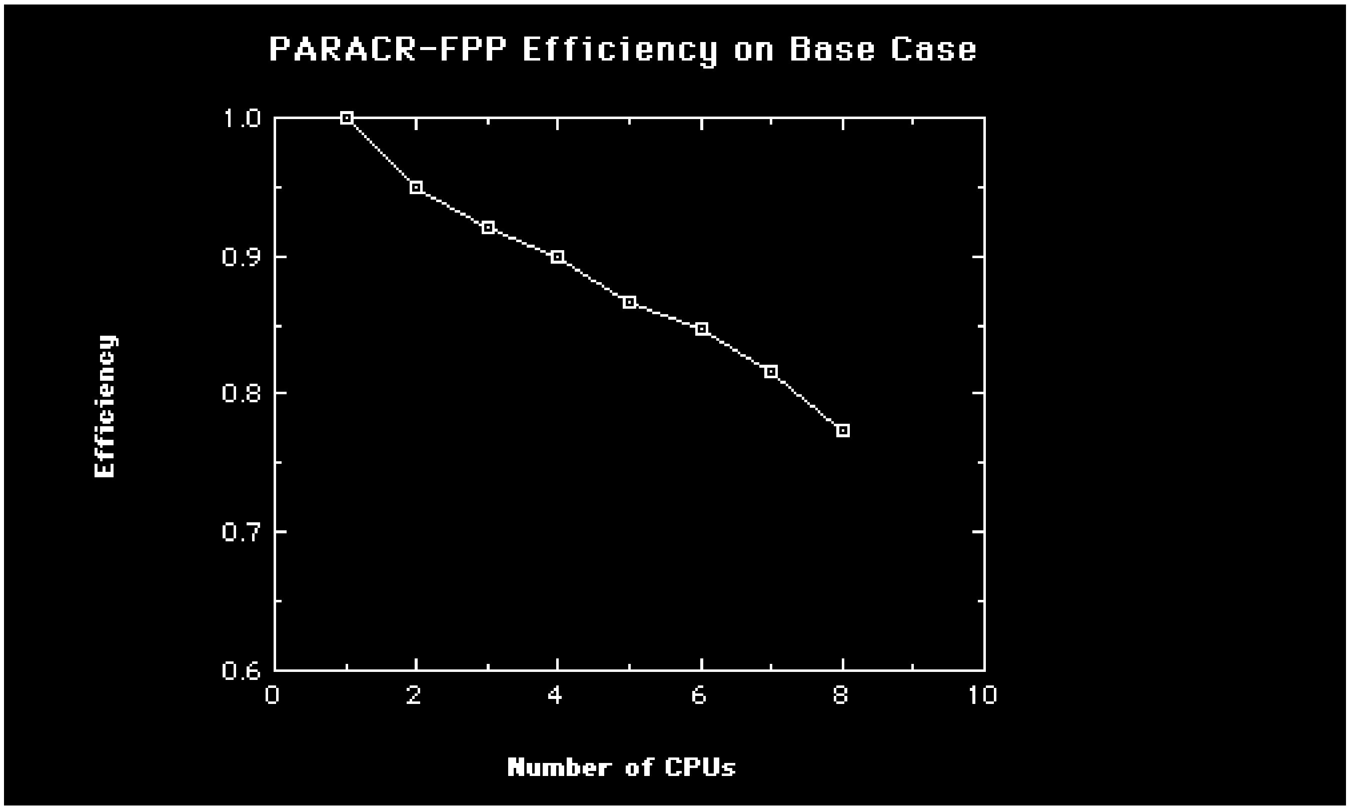
Table 5 shows parallel performance for the base case PARACR on increasing numbers of CPUs. Section 2.1 defines speedup, efficiency, and serial fraction.

Table 5
Parallel Y-MP Performance of Base Case (2**16) PARACR

NCPU	Elapsed Time	Speedup	Efficiency	Serial Fraction
1	16.807	1.000	1.000	1.000
2	8.843	1.901	0.950	0.052
3	6.085	2.762	0.921	0.043
4	4.672	3.597	0.899	0.037
5	3.877	4.335	0.867	0.038
6	3.304	5.087	0.848	0.036
7	2.941	5.715	0.816	0.037
8	2.719	6.181	0.773	0.042

Figure 2 shows that efficiency of the FPP-generated version of the PARACR algorithm decreases in gradual fashion as the number of CPUs increases. The excellent speedup and efficiency are somewhat surprising because this algorithm is memory-intensive: HPM measurements on the 8-CPU case revealed that the machine spent 42% of its clock periods holding instruction issue due to memory references. During such periods, the CPU could not issue an instruction because a memory port was busy or a resource such as a register was reserved by another instruction.

Figure 2



The key to parallel performance in this algorithm is the very long vector length. Since the system size of the base case (65536) is exactly divisible by 512, the parallel work can be divided evenly among the 8 CPUs while keeping the vector units of each CPU completely filled. The large number of instruction issue delays due to memory references do not present a severe bottleneck for this algorithm because computations can proceed in other parts of the CPU, such as the functional units.

4.3 FPP Evaluation

This section discusses the performance of FPP by showing, in Table 6, the speedup of the PARACR as a function of problem size. Section 3.3 defines the Table 6 column labels.

Table 6
Parallel 8-CPU Y-MP Performance of the PARACR

System Size	Pct Vector	Smax	Smeas	Pct Parallel
2**14	97.78	6.92	5.88	94.84
2**15	97.87	6.96	6.03	95.33
2**16	97.93	6.99	6.14	95.66
2**17	97.96	7.00	6.31	96.18
2**18	97.98	7.01	6.38	96.38
2**19	97.99	7.02	6.39	96.41

The difference between the maximum speedup based on percent of vector operations and the measured speedup diminishes gradually as the system size increases. The measured performance of the code increases only slightly with problem size since even the smaller cases have large vector lengths. The table indicates that FPP has extracted essentially all of the parallelism available, especially at the longer vector lengths. As noted previously, the clearing loop in the algorithm contains insufficient work for parallel partition by FPP, so this loop reduces parallel efficiency. The high degree of parallelism exhibited by this algorithm implies that a block cyclic solver may be more effective than the Gaussian solver in executing parallel solutions for two-dimensional NAS CFD codes.

5.0 Alternating Direction Implicit Algorithm

5.1 Code Description

The Alternating Direction Implicit (ADI) method forms an important class of solution procedures on the NAS machines, especially in the field of global circulation models (NASA, 1990). The ADI algorithm employs an operator splitting technique to decompose the problem into multiple one-dimensional subproblems. This decomposition increases the stability of the original problem and affords a larger timestep which, for most regular geometries, offsets the increased number of arithmetic operations produced by decoupling the original problem. The algorithm in the suite follows a standard implementation (Press, *et al.*, 1986), replacing the Gaussian solver described therein by a power-of-2 cyclic reduction method (Section 4.0) to solve the tridiagonal systems.

Similar to the SOR discussed in Section 3.0, the ADI code solves Poisson's equation with coefficient arrays describing the geometry. However, the SOR used a 3-dimensional cube whereas the ADI employs a 2-dimensional rectangular grid. The ADI would employ the same solution technique in three dimensions as it does in two dimensions. Features enhancing parallelism include the cyclic tridiagonal solver and the data parallelism of the algorithm. The problem geometry also contributes to the parallelism by allowing application of the same operations to points adjacent to the boundary. Termination occurs when the calculation has reduced the error as discussed in Section 3.

5.2 Code Performance

The base case treats a square with 1024 nodes per edge to give about 1 million points per grid. As with the SOR, the code contains arrays providing the geometry coefficients. The HPM reports a flop-to-memory access ratio of 0.95 and a vector length exceeding 63. The code executes the 4 MW base case in 135 seconds, performing at 130 MFLOPS. Single processor performance data for the ADI are shown in Table 7.

Table 7
Singletasked Y-MP Performance of the ADI

Size	MW	MFLOPS	CPU seconds
256	0.3	122.9	5.0
512	0.9	126.5	25.0
1024	3.4	129.8	135.0
2048	12.9	131.5	759.4

The table shows only a small increase in vector performance with system size. The overhead in the algorithm, notably the number of calls to the solver and the scalar indexing in the solver, increase with system size. Certain user optimizations such as inlining the solver and improving the solver indexing, would improve code performance.

FPP constructed a parallel version from the vector version described above by executing the filling and backfilling loops of the x-sweep and the y-sweep in parallel. Calculations in the cyclic tridiagonal solver are also performed in parallel. Synchronization of the processors occurs after the filling loop of the x-sweep, at the end of 3 DO-loops in the cyclic solver, after the backfilling loop of the x-sweep, after the filling loop of the y-sweep, again at the end of 3 DO-loops in the solver, and after the backfilling loop of the y-sweep. Thus, the parallel version requires 10 CPU synchronizations at the end of each plane, 10 times the number required by the SOR in Section 3. Even with the tridiagonal solver brought into the main routine, FPP could not recognize that each column of the x-sweep and each row of the y-sweep could execute in parallel. The tool exposed useful parallelism only in the solver.

Although the SOR and ADI versions in this suite solve different problems, other analyses (Cvetanovic, *et al.*, 1990) on equivalently optimized SOR and ADI algorithms have shown greater speedups for the SOR than the ADI; however, for most problems, the ADI algorithm will require less CPU time because it requires fewer iterations to achieve convergence.

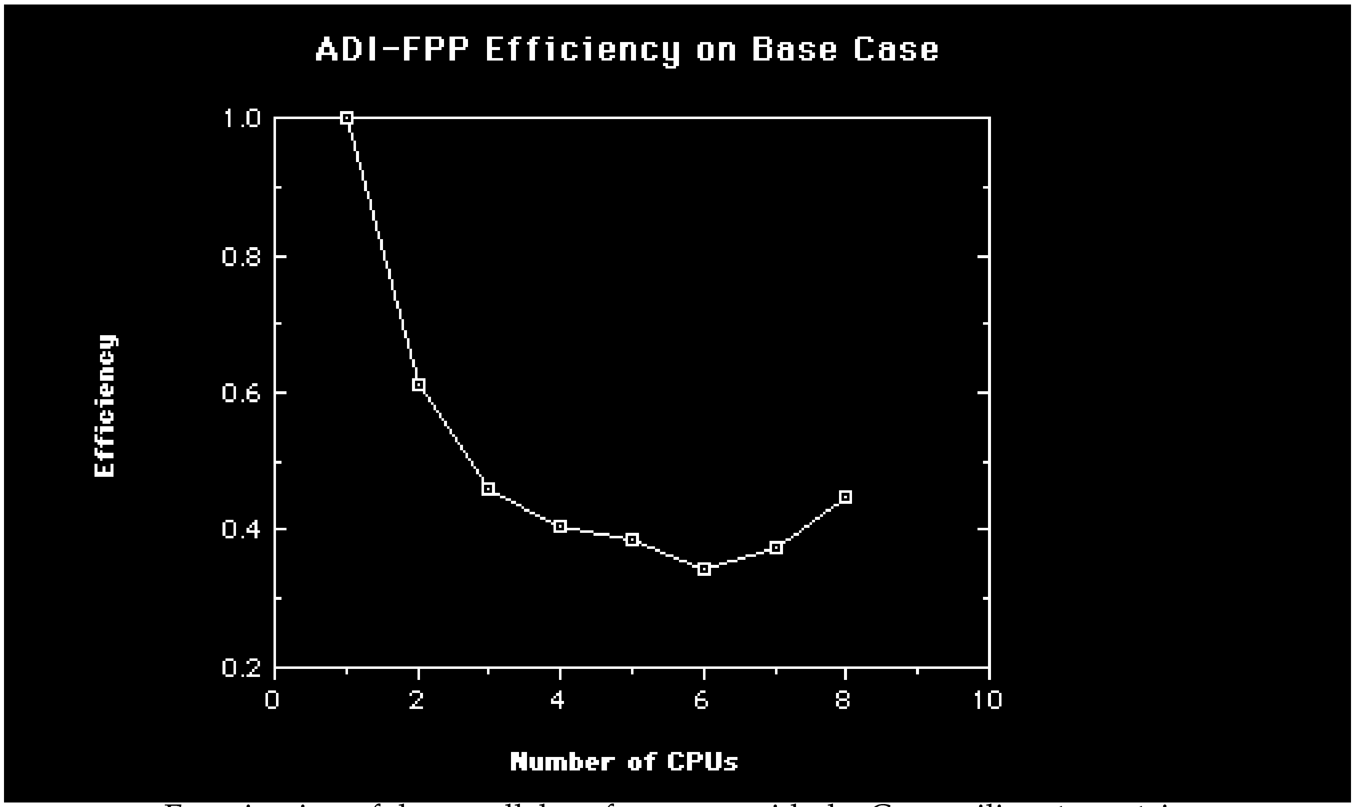
Table 8 shows base case parallel performance on increasing numbers of CPUs. Section 2.1 defines speedup, efficiency, and serial fraction.

Table 8
Parallel Y-MP Performance of the Base Case ADI (1024**2)

NCPU	Elapsed Time	Speedup	Efficiency	Serial Fraction
1	136.59	1.000	1.000	1.000
2	112.06	1.220	0.610	0.640
3	98.561	1.386	0.462	0.582
4	84.330	1.620	0.405	0.490
5	70.978	1.921	0.385	0.400
6	66.633	2.054	0.342	0.385
7	53.339	2.613	0.373	0.280
8	38.006	3.591	0.449	0.175

The following figure shows the efficiency of the FPP-generated version of the ADI algorithm to be lower than the previously discussed SOR algorithm and also lower than the tridiagonal solver used in the ADI. The efficiency of the ADI is much lower than would be expected if all vectorized floating point operations had been executed in parallel.

Figure 3



Examination of the parallel performance with the Cray utility *atexpert* indicated that insufficient work in certain parallel loops produced the poor utilization of the extra processors. These parallel loops filled the arrays for the tridiagonal solver and performed the backsubstitution after the solver. The extra CPUs require some time to arrive at the location in the code where parallel work is to be performed. If there is only a small amount of parallel work, the master CPU may have done it all, or maybe just one additional CPU is required. The fact that this ADI solves a particularly simple equation in only one variable exaggerates the parallel CPU utilization problem, but this difficulty has surfaced in previous analyses of CFD codes (Bergeron, 1992).

Despite considerable user intervention, FPP could not recognize that the x-sweep and the y-sweep could each execute in parallel. The tool uncovered useful parallelism only in the solver. The inability to expose higher level parallelism even on this simple code could be a serious drawback for machines with larger numbers of CPUs. For large numbers of processors, loops may contain an insufficient amount of work to utilize all available processors and a coarser-grained partition may be required to keep all processors busy. This inability may be especially harmful to the many CFD codes which utilize the ADI method to solve the Navier-Stokes equations.

The figure also shows that efficiency of the FPP-generated version of the ADI algorithm decreases sharply to a minimum at 6 CPUs and increases for

7 and 8 CPUs. The computational load was perfectly balanced for 2, 4, and 8 CPUs.

5.3 FPP Evaluation

This section discusses the performance of FPP by showing, in Table 9, the speedup of the ADI as a function of problem size. Section 3.3 defines the Table 9 column labels.

The table shows a decrease in the percentage of parallel code as the square size increases from 256 to 512. The increased size allowed satisfaction of a loop threshold test and FPP placed more DO-loops in parallel for the 512 square source. However, the overhead accompanying the parallel execution of these loops offset the decrease in elapsed time from multiple CPUs executing the actual calculations. The small amount of work contained in these DO-loops was essentially completed by the master task, and the utilization of the extra CPUs incurred synchronization overhead with little benefit. Section 2.2 indicated that the percent of parallel code was obtained from actual performance measurements. Since the measured parallel performance decreased, the amount of parallel code inferred from the performance measurement also decreased.

The table also shows a small decrease in percent vectorization and a large increase in the percent of parallel code as the square size doubles from 512 to 1024. The increased size allowed satisfaction of additional FPP loop threshold tests which resulted in parallelization of more DO-loops. Execution of this code in singletasked mode on one CPU requires execution of the additional scalar code accompanying the additional parallel loops and this additional scalar code produces the decrease in percent vectorization shown in Table 9. Parallel execution of these additional DO-loops combined with the execution of other parallel DO-loops (now made profitable by the increased square size) to increase the percentage of parallel code from 3% to 82%.

Table 9
Parallel 8-CPU Y-MP Performance of the ADI

Square Size	Pct Vector	Smax	Smeas	Pct Parallel
256	97.52	6.82	1.10	10.11
512	97.72	6.90	1.04	3.96
1024	96.60	6.46	3.59	82.49
2048	97.85	6.95	4.15	86.74

The difference between the maximum speedup based on percent of vector operations and the measured speedup diminishes somewhat as the system size increases. While FPP has partitioned most of the vector loops for parallel execution, loops with a small amount of work do not execute in parallel fashion. As described above, CPUs require a finite time for arrival and they may arrive after all work has completed. The measured performance of the code increases strongly with problem size primarily due to the increase in parallel work at the longer vector lengths. The table indicates that FPP has extracted about 87% of the parallelism available at the longer vector lengths. The short vector lengths also prevent efficient parallel performance of the cyclic reduction tridiagonal solver.

6.0 Multigrid Algorithm

6.1 Code Description

Multigrid solution techniques are an advanced form of relaxation algorithm which allows NAS users to accelerate the convergence of their CFD codes (NASA, 1990). Multigrid algorithms solve partial differential equations by employing relaxation methods to dampen high-frequency errors and by using multiple grids to allow an inexpensive solution to low frequency errors. A simple two-grid iteration technique begins by a number of relaxations on a fine grid followed by a projection of the errors to a coarse grid. Relaxations on the coarse grid are then followed by an interpolation back to the fine grid. Three basic operations comprise the multigrid technique: relaxation, projection, and interpolation.

The multigrid algorithm in the suite employs a 3-dimensional version of a standard V-algorithm (McCormick, 1987) to solve Poisson's equation on a power-of-2 cube. The base case employs 5 grids with 2**7 points on the fine grid and 2**3 points on the coarse grid. Consistent with the approach taken for the previous codes in the suite, this effort provided an efficiently vectorized algorithm as input to the automatic parallelization tool. The recursion present in a serial visit with a Gauss-Seidel solver to all grid nodes dictated a red/black ordering. A relaxation cycle includes separate visits to the red and the black nodes using separate Gauss-Seidel solvers. For each grid in the descending part of the V, the technique contains multiple relaxation cycles and a projection. For each grid in the ascending part of the V-cycle, the technique contains a relaxation on red nodes, multiple relaxation cycles and an interpolation from the coarse black nodes to the fine black nodes. Subroutines corresponding to the three multigrid operators, *i.e.*, relaxation, projection, and interpolation, consist of triple DO-loops visiting either the red or the black nodes.

6.2 Code Performance

The base case treats a cube with 129 nodes per edge, which corresponds to a problem size of about 1 million points. The singletasked version of the code executed the 7 MW base case in 31 seconds, performing at 127 MFLOPS. The autotasker was unable to parallelize the triply nested DO-loop of the Gauss-Seidel iteration because the red/black solver used the outer index to identify the red and black nodes on each plane of each grid. Full parallelism required manual replacement of the Gauss-Seidel iteration with a Jacobi iteration; this more parallel version required 40 CPU seconds due to the additional memory access and reflected a 25% performance degradation. The performance of the singletasked version of the MGR suffered because the base case displayed vector lengths of only 54.

Table 10 presents single processor performance data for the Jacobi iteration version.

Table 10

Singletasked Y-MP Performance of MGR

Cube Size	MW	MFLOPS	CPU seconds
33	1.6	35.6	1.6
65	5.3	64.1	7.5
129	7.2	99.7	39.8
257	56.0	115.0	280.0

The table shows a strong increase in vector performance with cube size. The cube sizes in the table overstate the vector length because the red/black solver divides the grid into two parts and halves the vector length. Moreover, the multigrid algorithm operates on cubes of various sizes, all of which are equal to, or smaller than, the cube size characterizing the problem. Thus, cube sizes of 65, 129, and 257 display hardware floating point vector lengths of 27, 54, and 63, respectively.

For the multigrid relaxation cycle described above, FPP constructed a parallel region by placing each iteration of the outer loop in parallel. The red iteration employs only black points, which are themselves constant during the red iteration. This technique decouples the calculation of the red planes and, for the black iteration, decouples the calculation of the black planes. For each color on a given grid, calculations on each of the planes are performed in parallel with FPP distributing one plane per processor. Synchronizations, amounting to nine per grid, occur after all planes have been updated, *i.e.*, at the end of the outer loop. Convergence checking occurs in singletasked mode because the program requested the maximum value of the error in addition to updating of the global sum used for convergence.

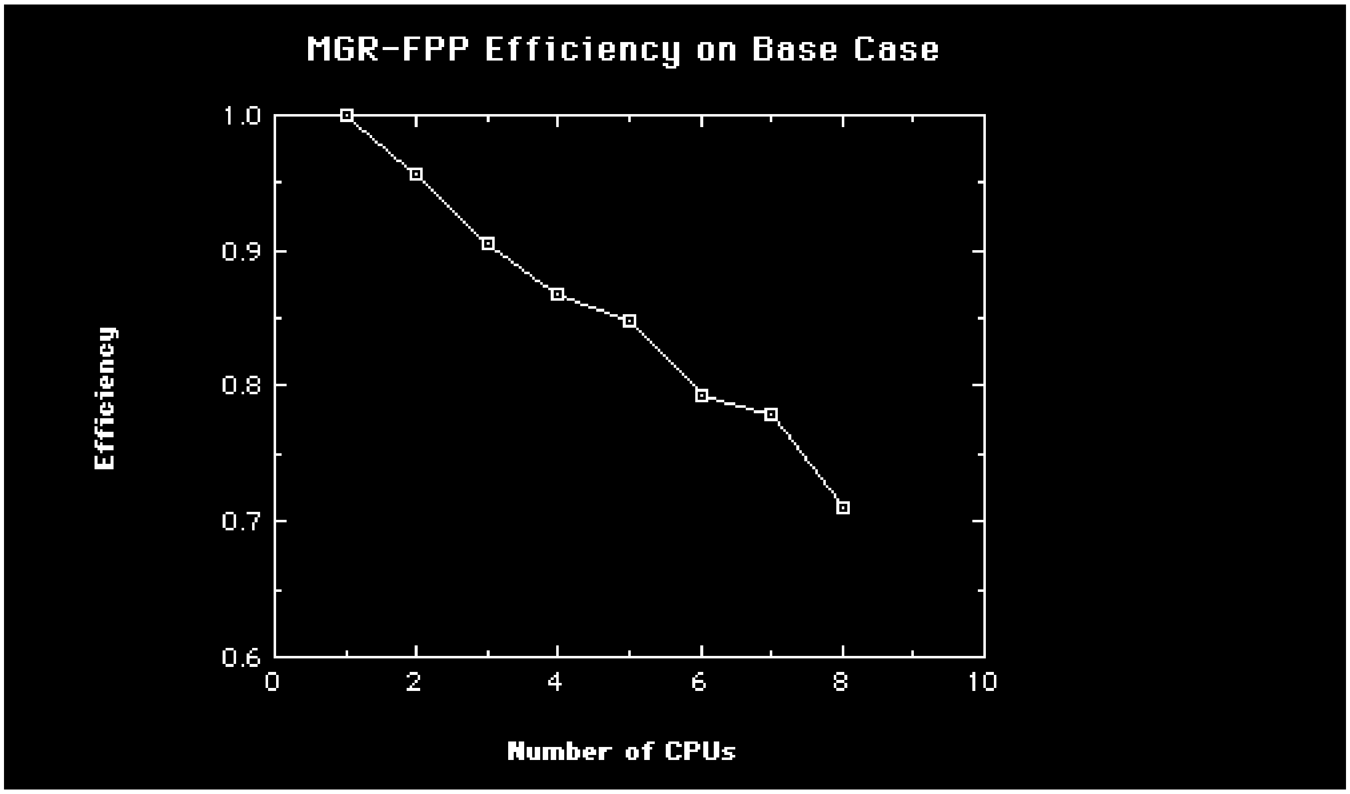
Table 11 shows base case parallel performance on increasing numbers of CPUs. Section 2.1 defines speedup, efficiency, and serial fraction.

Table 11
Parallel Y-MP Performance of the Base Case MGR

NCPU	Elapsed Time	Speedup	Efficiency	Serial Fraction
1	37.900	1.000	1.000	1.000
2	19.807	1.913	0.957	0.045
3	13.951	2.717	0.906	0.052
4	10.932	3.467	0.867	0.051
5	8.943	4.238	0.848	0.045
6	7.951	4.767	0.794	0.052
7	6.947	5.456	0.779	0.047
8	6.674	5.679	0.710	0.058

The table indicates an excellent speedup for the MGR algorithm. The magnitude of the speedup is surprising because the coarse grid vector lengths are quite small. Figure 4 shows that the efficiency of the FPP-generated version of the MGR algorithm decreases monotonically as the number of CPUs increases. The visiting of 5 different size grids by the three multigrid operators precludes an efficiency increase of the type observed for the ADI, which arises from a balanced computational load.

Figure 4



The short multigrid vector lengths make inefficient use of the vector floating point units and also produce a load imbalance because of insufficient work for multiple processors. The reason for the high efficiency is that the parallel version executes a large amount of scalar code in parallel. The scalar code arises from the integer offsets required to obtain the locations of the red and black nodes on the various grids comprising the multigrid scheme.

6.3 FPP Evaluation

This section discusses the performance of FPP by showing the speedup of the MGR as a function of problem size. Section 3.3 defines the Table 12 column labels.

Table 12
Parallel 8-CPU Y-MP Performance of MGR

Nodes/Edge	Pct Vector	Smax	Smeas	Pct Parallel
33	53.22	1.87	2.05	58.60

65	69.35	2.54	4.85	90.73
129	81.88	3.53	6.18	95.78
257	87.66	4.29	6.82	97.52

The table indicates that the FPP-extracted parallelism exceeds the vector-parallelism. The reason for this anomaly is that the large amount of scalar work spent in calculating node indices can be performed in parallel. The measured performance of the code increases dramatically with problem size due to longer vector lengths and smaller load imbalance effects.

The replacement of the Gauss-Seidel iteration with the Jacobi iteration, as required by full parallelism, negates the requirement for red/black ordering. Rewriting the algorithm with a standard Jacobi solver should lead to an improved convergence rate with longer vector lengths and greater parallelism.

7.0 Shallow Water Model

7.1 Code Description

Codes employing explicit timestepping of CFD equations are used by many NAS users, especially for treating rate-dependent phenomena, such as combustion and other chemical reactions (NASA, 1990). The finite difference equations for the shallow water model (SWM) employ explicit timestepping and space discretization based on Taylor expansions to represent computations employed in atmospheric modelling. The two-dimensional SWM equations are a system of three equations in three unknowns, the x-velocity, the y-velocity and the height of the fluid. The current formulation as presented by Sadourney (1975) and implemented by Hoffman, *et al.* (1986), conserves the mean square vorticity, or enstrophy. For stability in long-term atmospheric modelling, enstrophy seems to be a more important invariant than energy.

SWM employs a staggered leapfrog method to advance the conserved quantities in time. Double DO-loops sweeping over the entire grid contain the bulk of the calculation. The first double DO-loop spatially updates mass fluxes and the velocity potential; the second one integrates the mass and momentum equations; the final double DO-loop performs a time filtering for stability. Application of periodic boundary conditions occurs after each of the double DO-loops. Termination occurs upon completion of a specified number of timesteps.

Application of periodic boundary conditions and use of a regular geometry introduces a high degree of parallelism into the code. The shared memory architecture of the Y-MP facilitates implementation of these boundary conditions, but a distributed memory architecture would require additional message-passing to transfer data from one boundary node to another.

7.2 Code Performance

This section discusses the singletasked performance of the SWM and the parallel performance of the base case. Consistent with a 1 million point grid discussed in Section 1, the base case treats a square with 1024 nodes per side. The code executes the 15 MW base case in 304 seconds, performing at 222 MFLOPS. HPM data indicates vector lengths exceeding 63 and a flop-to-memory access ratio of 1.46, a ratio substantially greater than 1.0. The excellent SWM vector performance arises from the explicit nature of the time integration which calculates the new value at a point as the old value plus contributions from only the nearest neighbors. Since this algorithm visits the nodes in a sequential order, the CPU is able to store data temporarily in registers instead of fetching data from memory. Table 13 shows singletasked performance as a function of problem size.

Table 13

Singletasked Y-MP Performance of the SWM

Square Size	MW	MFLOPS	CPU seconds
128	0.3	217.7	4.8
256	1.0	221.3	18.9
512	3.8	222.5	75.6
1024	14.8	222.3	303.5
2048	58.9	222.0	1200.3

The table shows only a small increase in vector performance with increasing square size. As shown in Table 15, the fraction of floating point operations performed in vector mode increases very slowly with the increase in problem size. Since the vector units were filled at fairly small problem sizes, the performance remains constant with size.

FPP constructed a parallel version from the vector version described above by creating parallel regions for each of the 3 outer loops. Thus, the 8 CPUs could concurrently execute iterations of the first outer loop until all iterations were complete, then synchronize, and go on the next outer loop. The code required singletasked initialization of pressures and velocities and single CPU application of the periodic boundary conditions.

Table 14 shows base case parallel performance on increasing numbers of CPUs. Section 2.1 defines speedup, efficiency, and serial fraction.

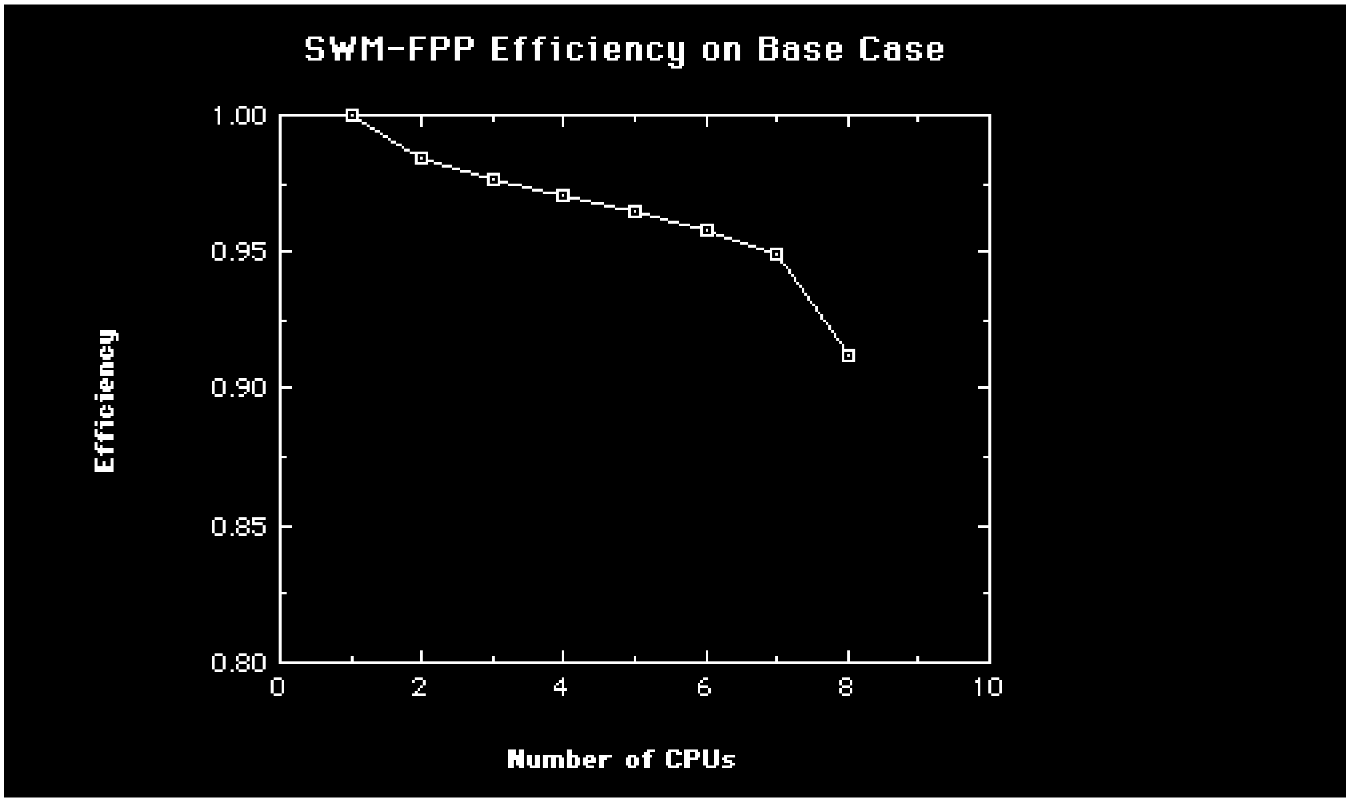
Table 14
Parallel Y-MP Performance of the Base Case (1024**2) SWM

NCPU	Elapsed Time	Speedup	Efficiency	Serial Fraction
1	296.000	1.010	1.010	1.000
2	152.000	1.967	0.984	0.017
3	102.000	2.931	0.977	0.012
4	77.000	3.883	0.971	0.010
5	62.000	4.823	0.965	0.009
6	52.000	5.750	0.958	0.009
7	45.000	6.644	0.949	0.009
8	41.000	7.293	0.912	0.014

Since the number of outer iterations is divisible by 64, the CPUs can execute an integral number of outer loop iterations, thus perfectly balancing the computational workload and keeping the vector pipelines filled.

Figure 5 shows that the efficiency of the FPP-generated version of the SWM algorithm decreases smoothly through 7 CPUs and then decreases sharply at 8 CPUs. According to *atexpert* , this decrease arises from a load imbalance generated by the singletasked regions. As the number of CPUs increase and the elapsed time decreases, this imbalance exerts a greater influence because the fraction of time spent in singletasked mode increases.

Figure 5



7.3 FPP Evaluation

This section discusses the performance of FPP by showing, in Table 15, the speedup of SWM as a function of problem size. Section 3.3 defines the Table 15 column labels.

Table 15
Parallel 8-CPU Y-MP Performance of the SWM

Square Size	Pct Vector	Smax	Smeas	Pct Parallel
128	97.12	6.67	3.28	79.45
256	97.39	6.76	6.57	96.90
512	97.51	6.81	7.69	99.42
1024	97.58	6.84	7.84	99.70
2048	97.61	6.85	7.95	99.92

The table indicates that the FPP-extracted parallelism exceeds the vector-parallelism. The reason for this anomaly is that the large amount of scalar work spent calculating node indices was also performed in parallel. The table indicates that this algorithm performs well at all problem sizes.

8.0 Discussion

The codes presented to the Cray Fortran PreProcessor (FPP) perform well on single CPUs of the Cray Y-MP. Since four of the codes (SWM is the exception) solve simple problems involving only a scalar field, the codes lack computationally rich DO-loops and would display higher MFLOP rates on problems involving vector-field solutions. However, all codes displayed reasonably long vector lengths and high fractions of vector operations. These characteristics presage favorable loop-level parallel performance because they indicate data independence and large amounts of parallel work.

Although FPP produced parallel versions of all vectorized codes with no user intervention, creation of efficient parallel programs did require some minor modifications to the algorithms. Since the codes were small, a simple review of the FPP output could reveal whether the preprocessor had created an efficient parallel program. This review involved simply scanning the outer computational loops for the construct "DO PARALLEL". If this construct preceded the main loop, then FPP had created a highly parallel region. An additional examination was required to verify that FPP had obtained the best possible parallel region.

In some cases, FPP generated these highly parallel regions, but execution on 8 CPUs did not lead to factor-of-eight speedups in these loops. The Cray utility *atexpert* was quite helpful in providing reasons for less-than-expected parallel performance. The following paragraphs discuss both the requirements of FPP for code rewriting and the role of *atexpert* in resolving performance questions.

8.1 FPP Performance

NAS users typically employ FPP to parallelize vectorized code involving explicit solvers, relaxation algorithms, and implicit solvers. The following discussion reflects lessons learned from the parallel suite with these three general algorithms.

Explicit Solvers

The SWM code used a standard explicit algorithm in which the new values at each node depend only on the old values and those of the nearest neighbors. SWM performs the same operations at all grid points; this procedure defines a data parallel algorithm. SWM singletasked regions involve initial conditions and the periodic boundary conditions.

Given a program with such high potential parallelism, FPP produced a program with high measurable parallelism. FPP required no vector source code modifications to cast all major DO-loops into parallel form. The singletasked regions did little to retard parallel performance, but *atexpert* indicated possible performance degradation with greater-than-8 CPU execution (Section 8.2).

Although practical CFD problems involve extra zones due to fluid property differences and extra terms arising from viscosity effects, explicit methods can treat all zones in similar fashion. The experience with the SWM algorithm indicates that FPP should generate efficient parallel code from those NAS programs employing explicit solution algorithms.

Relaxation Algorithms

The SOR and MGR vectorized iterative solvers employed red/black schemes, implemented in the case of the SOR with indirect addressing, and implemented in the case of MGR with indexing offsets.

Given the vector version of SOR, FPP was unable to construct parallel loops due to an index dependency. Replacement of the nested loop with a single loop containing a simpler indexing scheme allowed the generation of parallel outer DO-loops.

As with the SOR, an index dependency prevented FPP from constructing parallel loops for MGR. The use of SOR-like indirect addressing was possible, although MGR's use of five different relaxation grids would have required five different addressing arrays. Substitution of a Jacobi-like iteration allowed FPP to generate parallel code in the MGR solver.

In the case of the relaxation schemes with moderate to high potential parallelism, FPP was unable to produce a program with high measurable parallelism. These vectorized relaxation schemes required a few hours of additional effort to obtain more parallel solvers with FPP.

NAS user codes may have the same or similar problems because the logical decoupling required to generate vectorized relaxation algorithms does not always lead to the complete independence required for parallelism.

Implicit Solvers and Algorithms

Implicit schemes employ global solvers to obtain the exact solution of the system at a given timestep, and efficient parallel performance requires both a highly parallel solver and highly parallel pre-solver routines to assemble the left-hand and right-hand sides.

The PARACR code, exemplifying a shared memory global solver, employed a modification to the standard odd-even cyclic reduction to provide an algorithm with high potential parallelism. FPP recommended a reduction in the number of DO-loops to reduce the number of synchronizations. This modification required about 2 hours of user effort. Given a program with high potential parallelism, FPP produced a program with high measurable parallelism.

The ADI code combined a standard solution algorithm with the parallel cyclic reduction solver for the tridiagonal equations. FPP discovered useful parallelism only in the solver and could not recognize the parallelism in the x-sweeps and y-sweeps. While the inability of FPP to provide efficient parallel execution outside of the solver was due to computationally sparse DO-loops, its inability to parallelize the ADI at a higher level was more serious. Given a program with high potential parallelism, FPP produced a program with only moderate measurable parallelism.

Current NAS implicit CFD codes loosely couple the task of solving the system of equations to the task of assembling the left and right hand sides. Concerns for parallelism may not influence the data structures and their access patterns. For these codes, the current form of FPP may not be able to uncover parallelism at high levels. Since larger numbers of CPUs require higher fractions of parallel code to maintain the same level of efficiency, the inability to recognize parallelism at a higher level may limit FPP's future usefulness.

8.2 *ATEXPERT* Insights

FPP routinely generated highly parallel regions for codes in the suite, but wall clock measurements during dedicated Y-MP execution confirmed the *atexpert* predictions of less than factor-of-eight speedups in several of these regions. This section discusses several instances of the help provided by this tool. While in every instance, *atexpert* provided plausible reasons for performance anomalies, the tool required the user to review its output quite carefully. A cursory examination of results with poorly conceived questions provided little insight.

In the ADI code, *atexpert* indicated that the extra CPUs required a significant amount of time to traverse the various "if" statements between the top of a subroutine and the parallel work. Since the master CPU will be working while the extra CPUs are finding their way to the parallel work, parallel loops must contain a significant amount of work or else the master will have reduced the parallel work available for the extra CPUs.

SWM measured efficiency displayed a sudden downturn as the number of CPUs increased to 8. *Atexpert* indicated the reason was the growing influence of the singletasked periodic boundary conditions.

Atexpert indicated memory conflicts would degrade the parallel performance of the relaxation schemes, but since the arrays were already carefully dimensioned, there seemed to be little improvement possible in these routines.

The overall experience with *atexpert* was quite pleasing, and its clear presentation format represents the state of the art in this area. The tool could profit from the addition of code to supply the user with performance-improving suggestions. For example, in ADI, *atexpert* could recommend more modular coding to allow CPUs to enter parallel regions at the top of the subroutine. In SWM, *atexpert* could suggest a rewriting of the boundary condition update to maximize parallelism.

9.0 Conclusions

Although supercomputers increasingly employ multiple CPUs to provide greater computing power, the burden of harnessing this power through parallel processing lies with the programmer. Automatic parallelization tools promise to make the programmer's task easier, but the tools must be able to discover a wide range of parallel constructs. This report has described a suite of codes for testing automatic parallelization and has discussed the ability of Cray's loop-level parallelizer, FPP, to generate measurably parallel code.

The codes in the suite provide a good basis to evaluate automatic parallelization tools because they permit a large amount of parallel execution and because they contain several levels of parallelism. Most of the codes can perform efficiently with loop-level parallelism, but at least two of the codes, the SOR and the ADI, can execute in parallel at a higher level. The simplicity of the codes and utilization of well-known numerical methods allow higher levels of parallelism, either manually by user modification or automatically by software preprocessors.

The Cray automatic parallelization tool, FPP, exploited much of the loop-level parallelism in the codes comprising the test suite. Dependencies, introduced by the use of indirect addressing for vectorization, prevented generation of parallel code for some of the solvers. For the codes in the suite, FPP was unable to diagnose parallelism at a level higher than a DO-loop and Cray should place more emphasis on this aspect of parallelism. FPP performs a valuable, supplementary role as a worksaver. However, the effectiveness of the tool is proportional to the skill of the analyst.

Cray also provided an insightful parallel performance evaluation tool, *atexpert*, to complement FPP. In every case, this tool provided plausible reasons for parallel performance deficiencies. Use of *atexpert* should almost certainly increase the skill of the analyst. Currently available performance tools for distributed memory machines provide much data and little insight. NAS could make a contribution in this area by creating software which would diagnose performance problems on distributed memory machines.

For the parallel suite, future work will involve first the manual transformation of the codes into sources suitable for execution on a highly parallel machine. This effort may well involve more than loop-level transformations. The experience of manually transforming the codes will enable an informed evaluation of automatic tools on highly parallel machines. Future work will also involve porting these codes to the C-90. This effort should clarify whether loop-level parallelization can operate effectively on systems with moderate numbers of CPUs.

10.0 References

Adams, L., and Ortega, J. M., 1982. “A multi-color SOR method for Parallel Computation”, in *Proceedings of the 1982 Conference on Parallel Processing,*, pp. 53-56.

Bailey, D. H., et al. 1991. “The NAS Parallel Benchmarks-Summaries and Preliminary Results,” in *Proceedings of the Supercomputing ‘91 Conference*, pp. 158-165.

Bailey, D. H., 1992. “Misleading Performance in the Supercomputing Field,” in *Proceedings of the Supercomputing ‘92 Conference*, pp. 155-158.

Bergeron, R. J., 1992. “Autotasking STAGE-2,” NAS Technical Report RND 92-014, Ames Research Center, Moffett Field CA, 1992.

Buzbee, B.L., Golub, G.H., and Nielson, C. W., 1970. “On direct methods for solving Poisson’s eqations,” *SIAM Journal of Numerical Analysis* 7 pp. 627-656.

Chen, D., and Pase, D., 1991. “An Evaluation of Automatic and Interactive Parallel Programming Tools,” in *Proceedings of the Supercomputing ‘91 Conference*, pp. 412-423.

Cray Research Inc. 1988. *Cray Y-MP and Cray X-MP Multitasking Programmer’s Manual*, Pub. No. SR-022E, Cray Research Inc.,1988.

Cray Research Inc. 1990. *UNICOS Performance Utilities Reference Manual*, Pub. No. SR-2040 6.0, Cray Research Inc.,1990.

Cray Research Inc. 1991. *CF77 Compiling System, Volume 4: Parallel Processing Guide,* Pub. No. SG-3074 5.0, Cray Research Inc.,1991.

Cvetanovic, Z., Freedman, C. G., Nofsinger, C. 1990.“Efficient Decomposition and Performance of Parallel PDE, FFT, Monte Carlo Simulations, Simplex, and Sparse Solvers,” in *Proceedings of the Supercomputing ‘90 Conference*, pp. 455-464.

Hockney, R. and Jesshope, C. 1988. *Parallel Computers 2.*,Bristol, England: Adam Hilger Ltd.